Pixon Developer's Guide

Sandy Antunes

last updated: June 13, 2007

To Do list

March 22, 2007

John's priority rankings (*s*=stable, *t*=testing, *o*=ongoing work, *I*=incomplete)

s 1.1) ingest FITS from SECCHI, SOHO, RTWL (plus masks, etc)
s 1.2) true noise model
s 1.3) printed output and summary
o 1.x) ease of use
s 2.x) on-screen output and visuals
o 2.x) runtime for Pixon
o 3.x) resolution (assoc)
t 3.x) tests of geometry
t 3.x) interoperability with RTWL
t 3.x) F-corona brightness model subtraction
I 4.x) feature evolution vs misalignment investigation
I 4.x) run Pixon test problem
I 5.x) demo to others

Current installation of Pixon does not have a valid 2007 license but does use a temporary saved object provided by Amos. Sandy is currently running with a modified set of the Pixon source for exploring higher resolution runs and for profiling, available only on duffer.

Sample CMEs to explore:

LASCO Lightbulb: 13 Oct 97, C23, 11:26:05, 11:50:05, 12:06:05, 12:26:05
1st SECCHI CME: Jan 28-29, or try 12/30/06 06:00 or Jan 14-15th in HI and Cor2

Paul Notes

"Note: Memory requirements exceed 1GB for N>64" [tpixon] ">128" [cpixon]

CPixon times $T=0.0016n^3$ minutes, TPixon $0.0036n^4$ minutes.
Problem was H matrix with N^5 elements

June 2003: Paul and Amos both max out at N<203 (Paul=unix, Amos=Win2k, NT)

Pixon a priori assumptions: positivity, smoothness, and simplicity

Paul's 3 obstacles to Pixon's success:

1) Can we incorporate a priori knowledge
2) Can we do it in time? Can we get Amos to write code? "Never"
3) Will Pixon ever run fast enough to be usable?

User Guide

Required from user:

* FList, a list of input FITS filenames for SECCHI, LASCO or simulated data
* If using cropped data, the file that contains the cropping metadata (new image centers, sizes, and resulting sun center positions)
* Ncube, the resolution of the run: 16, 32, 64, 96, 128, 192, 256
* Max_Runtime, maximum number of hours the run should take
* gname, the short descriptive name for this run
* note, any longer description desired

Optional science values:

* n0, the electron density per cc, defaults to 1, often 1e8 is a good value too.
* chl, the desired limb darkening model, default is '2' aka extended sun with polynomial limb darkening:
    0 = point sun, no limb darkening
    1 = point sun, polynomial limb darkening
    2 = extended sun, polynomial limb darkening
* imodel, instrument model for get_Istd, defaults to '0' aka the Paris/Palo Alto case:
   0 = 'Paris/Palo Alto', Istd = K/10 * Im/ImIO * 555.0e-7 / (h*c)
   1 = '7000A center', Istd = K * total[0.2e(-(w-7000)^2/2var^2) * Iw) * dw

Optional performance values:

* autorange, sets the desired internal scaling, forcing max(data) to autorange. Default is to not rescale data. A good value is autorange=1000. This is the only 'optional' value that usually should be set to something other than its default. We only default it to 'off' because it enforces data scaling.
* noshm, toggle for whether to use shared memory fully, partially, or not at all. Defaults to yes.
* npo and noct, number of PIXON kernels per octave and number of octaves, together sets the total number of PIXON kernels npo*noct + 1, defaults to 2 per octave and 4 octaves (implying 9 kernels).
* interp, the desired renderer pixel interpolation scheme, default is '5' aka trapezoidal. Both native IDL and C versions exist; the C versions require that you have the compiled libraries installed and in your path.
    0 = none, coded in native IDL

1 = yes, v=1, coded in native IDL
2 = none, coded in C
3 = yes, v=1, coded in C
4 = yes, v=variable, C, requires 'psp' variable for data psf for interpolation
5 = yes, trapezoid spreading, C

* abstol, Pixon cutoff absolute change for fit, defaults to 1e-5.
* reltol, Pixon cutoff relative change for fit, defaults to 1e-5.
* annealing options: anneal=1, optional jsched/msched/rsched for annealing, see annealing docs.

Derived from FITS, no user input needed: All the geometry (in HAE reference frame), data scaling, instrument types, polarizations, noise, masks, and binnings are handled by the various ingest and wrapper programs.

Pixon Flow

Sandy Antunes
April 2, 2007

Pixon Usage and Processing, Short Form

1) Gather data and decide on resolution
    Get list of data files with festival/scclister/etc to make 'filelist'
    Decide on desired resolution of simulation 'N'

2) *p_process_filelist,N,filelist,pxwrapper,oPixon*

3) Run Pixon with *p_pixonloop*, visualize with *p_report*

Pixon Usage and Processing, Long Form

I) Gather data and decide on resolution
    Get list of data files with festival/scclister/etc to make 'filelist'
    Decide on desired resolution of simulation 'N'

II) Initialize structures:
    run "*prep_3drv,pxwrapper,views=n_elements(filelist),N=N*"
  (If you skip this, '*p_process*' will do it but you cannot tweak
   kernel parameters if you use '*p_process*' to make it.)

III) Optional, tweak kernel parameters for performance with *p_tweak_kernel*

IV) "*p_process,filelist,N,pxwrapper,oPixon*"

  i) create placeholders for data, noise, masks as N,N,nfiles arrays
    Also creates 'pxwrapper' if pxwrapper does not yet exist.

  ii) Noise generation:

    A) Have Level 0.5 data?
      generate noise from 0.5 using noise methods 1, 2 or 3 (1 is best)
    C) Have only Level 1.0 data?

generate noise from 1.0 using noise methods 4 or 5 (4 is best)

iii) Data check:

A) Have Level 1.0 data?

move to next step

B) Have only Level 0.5 data?

run *secchi_prep* on 0.5 to make level 1.0 data

iv) Data ingest:

run *p_ingest_fits* on level 1.0 data

v) Structure updating:

insert srjr from *p_ingest_fits* into pxwrapper.sr

("*p_update_struct,pxwrapper,'sr',srjr,iele*")

(can also be done in *p_ingest_fits* with pxwrapper=pxwrapper,iele=iele)

vi) Data collection:

accumulate data, mask, noise sets

vii) (repeat ii-vi for each additional data file)

viii) Prepare Pixon object and insert data items:

"*p_generate_pixon,'classic',pxwrapper,oPixon*'

"*p_insert_data,pxwrapper,data,oPixon,sigma=noise,mask=masks*"

VI) Run Pixon with *p_pixonloop*

VII) View and visualize results with *p_report* or other tools.

VIII) Write scientific paper and submit to journal.

Pixon Underlying Code

Notes on data units and normalizations.

- In terms of Paul's rendering, data is presumed to be in photons/sec.
- We presume here that 1 photon equals 1 electron.
- He provides a scaling term 'n0' in units of electrons/cc, but the implementation does not actually affect the rendering or reconstruction. Keep as n0=1.
- If you use 'p_ingest_fits', that routine converts MSB etc into photons/sec.

Make sure that, for longer exposures, you divide the data so it is photons/sec, not just photons. The noise calculation must also match the data scaling.

The value ‘Istd’ is used to multiply (normalized) data to get the intensity at the detector. It uses n0 (as well as other things) to produce a final value of photons/sec/cm^2/sr/A.

Polarizations in FITS are:
Polar: /polariz_on -> total brightness
1001 - total brightness 1002 - polarized brightness
1003 - percent polarized 1004 - polarization angle

Geometry notes:

Pixon uses a normalized scale where 1=1RSun

It requires (for cpixon) rho/gamma, LL, and z0 plus eps (pointing angle from suncenter)

It includes a conversion factor ‘L0’ of 1AU in solar radii.

The plate scale is L0tau (in AU) or tau (in Rsun), depending on which units a given routine needs:

o_flimb uses L0tau
get_istd_nd uses L0tau

Centers:

A simple example of a SECCHI FITS data file with N=4, aka 4x4 image, uses a start index of 1 with end index of N and is shown in Figure Ci.

Its coordinate frame uses the center of pixels, e.g. the point at *(2,2)* is *x*, at *(2.5,2)* is *y*, at *(2.5, 2.5)* is *z* (where we note in the picture that *z* ends up being our exact image center), as shown in Figure Cii.

In IDL, arrays go from 0 to N-1, as shown in Figure Ciii.

Pixon converts 'indices' to 'reference coordinates' as follows, where we will calculate where our above *x,y,z* are in Pixon pixels. The default Pixon 'center' is 0.5 * (N-1) * [1,1,1]. E.g. in 4x4 it is pixel (1.5,1.5) as shown in Figure Civ.

Thus the pixon reference point is the lower left corner of the desired position.

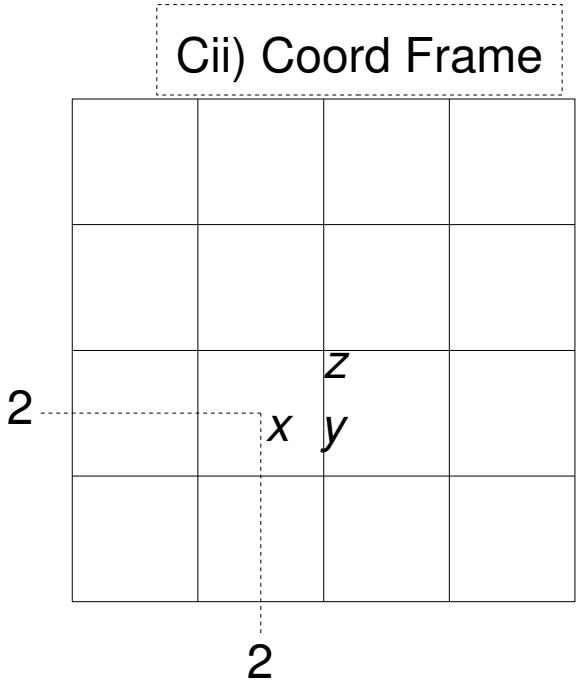
So... the FITS equivalent of "the center" *(2.5,2.5)* (aka *z*) has the Pixon pixel coordinates of (1.5,1.5). Therefore, FITS center values for the optical center CRPIX (from which we get our Pixon sr.k0 optical center) yield our Pixon sr.k0=(CRPIX-1,CRPIX-1). And, the HAE sun center (in the center of our image cube) is just our 0.5 * (N-1) * [1,1,1].

Center Pixel Reference Diagram

Ci) Data 4x4

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2	2,2	3,2	4,2
1,1	2,1	3,1	4,1

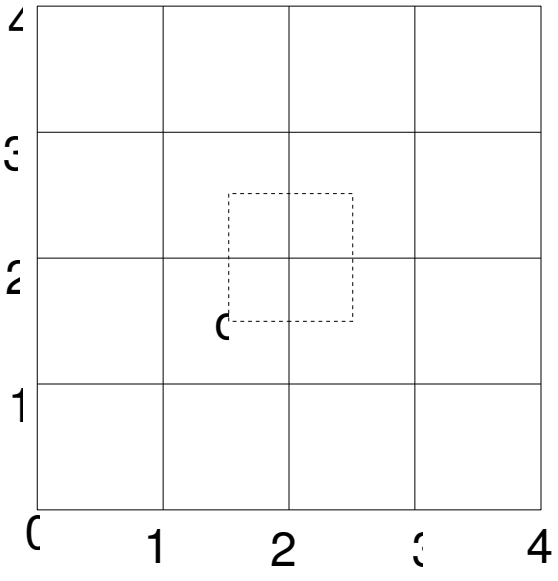
Cii) Coord Frame



Ciii) IDL 4x4

0,3	1,3	2,3	3,3
0,2	1,2	2,2	3,2
0,1	1,1	2,1	3,1
0,0	1,0	2,0	3,0

Civ) Pixon



HAE:

We use HAE (ecliptic rotational frame) x,y,z converted from km to AU.

The reason we keep HAE as our x/y/z is that HAE allows us to compare across models at different times, because it is an absolute reference frame.

So we derive

```
rho=atan(y,x)
phi=atan(z,sqrt(x^2+y^2))
LL=sqrt(x^2+y^2+z^2)
z0=LL * sin(phi)
```

LL is usually multiplied by L0 to put into units of Rsun

ffunc uses x,y,rho

'userstruct' for the renderer stores lla0, rho, d^2, tau

Routines

prep_3drv,pxwrapper,views=nv,N=N,enorm=enorm

* calls:
  p_build_sr, *p_build_sp*,*p_build_sn*,*p_build_sc* and makes 'pxwrapper'
* Optional values to supply are: model, ftol, enorm, gamma (simplified)
  Default normalization enorm is n0=1
* Most crucial is: p_build_sr:
  sets sun at center of image cube, sr.lla0
  sets L0 as 1 au in solar radii (our unit scaling), d=1
  sets default k0 values as data centers
  sets default polarity as 'b'
  sets default instr model as Cor2 (tau=60/N/L0, L0tau=60/N, imodel=1)
  imodel is used later in *get_istd_Nd()*

[*p_tweak_kernel(pxwrapper.sp,noct,npo)*]

* Used to alter the kernel variables to allow for larger runs

p_ingest_fits,fname,returndata,mask,srjr,[N=N]

* does _not_ use pxwrapper or oPixon object, is stand-alone and generic.
* reads fits headers and also gets data
* if given an N=N that does not equal the data size, rebins
  the data conserving total signal
* sets L0=214.943, aka 1 au in solar radii
* keeps default 'd' as length of voxel edge in AU def 1.0?
* keeps sun-center in image
* sets L0tau aka plate scale as per instrument, either hardcoded
  by instrument name or, if 'rsun' is in FITS header, as
  pixel 'cdelta1' (assumed in arcsec) over 'rsun' (assumed in arcsec),
  L0tau = instrument_FOV_Rsun / N
  FOVs defined include: Cor1, 4; Cor2, 15; EUVI, 1; C2, 12; C3, 60,
* sets tau=0.5*L0tau/L0
* gets WCS coordinates in HAE (Heliocentric Aries Ecliptic) x/y/z
  and converts them from HAE's km to AU.
* derives spherical rho, phi, LL, zz from HAE:
  rho = atan(y,x)
  phi = atan(z, sqrt(x^2+y^2))
  LL = sqrt(x^2+y^2+z^2)
  z0 = LL * sin(phi) ; distance above x-y plane
* gets polarity: so far 'b', 'p', plus filter angles if 'p',
  though dsf/dsft can handle 'b','p','t','r'
* extracts actual optical center and sun center
* gets masks for occulters using *p_getmask()* using N and L0tau
  along with hard-coded inner and outer radii (plus optional 'ratio')
  Cor1: 1.4-4.0, Cor2: 2.0-15.0, C1: 1.1-3.0, C2: 1.5-6.0, C3:3.5-30.0
  Masks are assumed centered on the optical center (not sun center)
* sets Istd=1.0 and imodel=0 as to-be-discarded defaults.
  Istd is what you multiply norm data to get intensity at detector
  in photons/sec/cm^2/sr/A. Note this is just a placeholder as
  p_generate_pixon() will do the proper intensity call via *p_osetup*.
* gets msb2dn and dn2photon values, guesses otherwise msb2dn=1, dn2photon=15
* rescales the data by 'n0'
* builds 'srjr' structure and returns it, returndata, and mask.
* User must make sure to insert 'srjr' into pxwrapper
  via 'update_struct_arrays,sr,srjr,iele' before calling p_generate_pixon,
  and put in 'returndata' and 'mask' via p_insert_data later

p_generate_pixon,'classic',pxwrapper,oPixon,image=timage,raw=raw

* resets pxwrapper.sr.lla0 to center of cube
* calls *pixon_env,'classic'*
  sets commons, paths, envs, loads binaries and license,
  forces compile of Paul's pxndsf and pxndsft
* calls *p_center_data,pxwrapper.sr* and shifts pxwrapper.sr.k0 by 0.25 (why?)
* normalizes timage by n0:rsync
  e.g. if n0=1e9 and the image is a model ranging from 0-1e9,
  the image gets properly rescaled to 0-1. You can use the /raw
  flag to override this behavior, but rarely should you need to use that.
* calls *p_osetup,pxwrapper,timage,oPixon*
  calculates std intensity via *o_flimb()* and *get_istd_Nd()*

rsync

this uses imodel, choice is
0: wl=555e-7 cm, Istd=K*0.1*(Im/Im10) erg/sec/cm^2/sr/A,
then convert Istd to photons/sec/cm^2/sr/A by wl/(h*c)
1: FWHM = 1000, w0 (central wavelength)=7000, Q0=0.2,
Istd=K * total(Q*Iw)*dw
Note that the global sr.n0 sets the proportionality constant K
Also uses (L0tau^3/L0^2) * cm/AU

The data n0 _must_ be the same as the global sr.n0, if not that has to be handled at the data level (as it is with p_ingest_fits). Since we use a global n0, we do not handle that anywhere else. Right now, this (*p_generate_pixon*) is the only time we set these values. Note that, if n0=1, you may get lucky even if you forget to rescale, but do make sure to check n0 if using ingest routines other than p_ingest_fits.

gets prf2d via *gprf()*
calls *ffunc()* and *PrepFPRF()*

sets a default mask (weight, wt) of value 1 (uniform weight)
sets a default 'sigma' image of sn.ftol[0] (sigma=0 makes Pixon fail)

creates Paul's “userstruct” which contains all the necessary geometry. Geometry is only handled by his DSF/DSFT tools, therefore the Pixon object itself contains no specific geometry.

finally, instantiates oPixon object (with dummy data arrays)

p_insert_data,pxwrapper,datum,oPixon.sigma=sigma,mask=mask
  * inserts the data into oPixon object, which must already be defined
    for that size data! Does no other changes (no n0 or geometry updates)
  * Also puts in 'sigma' noise, if given, and mask weights 'mask', if given

p_pixonloop,pxwrapper,oPixon,[/init],[/full,/pseudo,/anneal],[/flush]
  * inits, solves, plots, tallies
  * '/init' runs oPixon->Full,/restart for 1 step to make blank maps
  * '/flush' removes any existing solution image and puts in a uniform cube

p_report,pxwrapper,oPixon,/plot,/echo,/save
  * writes out runtime report, optionally plots and saves pxwrapper and oPixon

[datum=*stereo_project(pxwrapper,oPixon)*]
  * renders image into data as per the geometry defined in the structures.
  * is equivalent, with more formalism, to:
      for m=0,Nd-1 do data[*,*,m] = oData[m]->dsf(image)

Thompson scattering rendering DSF and DSFT routines are in pxndsf_force.pro and pxndsft_force.pro.

Noise Methods

Routines

```
; now coded as p_noisewrapper.pro

; Method 1: use secchi_prep directly, likely more accurate and consistent
;  noise = sigmaMSB = secchi_prep{sigmaDN + offsetbias}
  noise_method1 = secchi_noiseprep(data_L05, hdr05, /subtractbias, /cleanup)

; Method 2: make noise using level 0.5 data and fractional error
  fractional_error = fractional_noise(data_L05, hdr=hdr05, /subtractbias)
  noise_method2 = data_L10 * fractional_error

; Method 3, like Method 2 but not invoking secchi_prep
; noise_method3 = ( (noisedata - hdrn.biasmean) * dn2msb / exptime) * data_flat
  sigmaDN = fractional_noise(data_L05, hdr=hdr05, /subtractbias, /sigma)
  noise_method3 = fake_secchi_prep(sigmaDN, hdrn, /use_calimg)

; Method 4, if we only have Level 1 data, using Method 2 (fractional error)

  fakehdr={SECCHI_HDR_STRUCTNOISE,$
            BIASMEAN: 0, $
            EXPTIME: sr.exptime, $
            DETECTOR: sr.detector, $
            OBSRVTRY: sc.sat_name, $
               IPSUM: 0 $
           }
  data05=secchi_unprep(dataL10,fakehdr)
  fractional_error = fractional_noise(data_L05)
  noise_method2 = data_L10 * fractional_error

; Method 5, if we only have Level 1 data, using Method 3 (fake prep of DN)

  fakehdr={SECCHI_HDR_STRUCTNOISE,$
            BIASMEAN: 0, $
            EXPTIME: sr.exptime, $
```

```
            DETECTOR: sr.detector, $
            OBSRVTRY: sc.sat_name, $
               IPSUM: 0 $
            }
 data05=secchi_unprep(dataL10,fakehdr)
 sigmaDN = fractional_noise(data_L05, /sigma)
 noise_method3 = fake_secchi_prep(sigmaDN, fakehdr, /use_calimg)
```

Header requirements

fractional_noise
 no real hdr needed- only requires 'biasmean' and even defaults to 0 if no hdr,

secchi_unprep & *fake_secchi_prep*
 faked header okay-- both require hdr 'biasmean' plus call *get_calfac* to use
 'exptime','detector','obsrvtry','ipsum', also can call *get_calimg*
 which uses many unsaved header values, so do not use /use_calimg
 if you are giving it a faked header

secchi_noiseprep
 requires a full secchi hdr structure for *save_secchi05*
 because that goes into *secchi_prep*

Levels of Abstraction in Pixon

Sandy Antunes
April 2, 2007

DATA REPRESENTATIONS

COMMONS: 'debugs'

Paul's code has a COMMON block named 'debugs'. We have kept this in for compatibility with earlier routines of his.

STRUCTURES: 'pxwrapper'

Our implementation uses Paul's geometry and renderer, which has a series of structures called 'sr', 'sp', 'sc', and 'sn'. We bundle these together into a wrapper structured called 'pxwrapper'. These items (pxwrapper.sr, pxwrapper.sc, pxwrapper.sn, pxwrapper.sp) contain all of the user-choosable parameters defining a given problem.

USERSTRUCT: 'userstruct'

Paul's code has a structure called 'userstruct' which is derived from the 'pxwrapper' entities and placed into the Pixon Object. A copy is kept in the pxwrapper structure as well. This is used by the dsf and dsft IDL and C code.

DATA: e.g. *.fts, *.sav

We have the observed data sets, their associated masks (if any), and their calculated noise. These three items-- data, mask, noise-- all each 2D IDL arrays. We typically collect them into N x N x Nviews arrays.

There is also the image cube of size N^3, which is the underlying electron density. This is the solution Pixon produces. You can also specify a specific underlying image for rendering or as an initial guess at a solution.

OBJECT: 'oPixon'

PIXON uses an object representation to do most everything. It includes all data, masks,

images, noises, the userstruct, matrices and similar 3drv data items, with geometry as specified in the pxwrapper structure. Upon solving, it also contains the residuals, chisq of the fit, and other data for evaluating the result.

SOURCE CODE

SSW:

We use many SSW utility routines, so SSW must be loaded.

PIXON SOURCE CODE:

Pixon has its own source tree. It gets put into the path via the '*pixon_env*' routine (also called in '*p_generate_pixon*'). We track a system variable, '!pixontype_loaded' in order to avoid rebuilding the path multiple times.

There are two Pixon trees: Classic cartersian Pixon, and Tetrahedral Pixon. Both environments can be loaded but they have different underlying objects. The Pixon object from one cannot be operated on by the software for the other.

RENDERERS: *pxndsf*, disum*, dsum**

We have Paul's dsf and dsft routines, in both IDL and in C. The C routines are faster (about 10x faster than the IDL ones), but the C versions are not compiled for a given machine, the IDL ones are available.

ATOMS: various names

Most of Paul's code is used to set up the geometry and physics of the problem. We use these along with other short functional routines, dubbed 'atoms'

ROUTINES: *p_** and others

We have written wrappers around Pixon and Paul's code in order to handle input and output and proper problem set-up. These include routine tasks ranging from 'prepare default structures' through 'ingest FITS files' and 'calculate noise' to 'run Pixon in a loop for a set number of hours'.

SCRIPTS: *p_**

We have higher level scripts that call the proper routines in the proper sequence for solving real problems and visualizing results.

At the highest level, Pixon can be run with just 4 scripts: some sort of file selection tool, '*p_process_filelist', 'p_pixonloop*', and '*p_report*'

WIDGETS: *s3drv_**

We have or are creating widgets for graphically running our Scripts and Routines.

GUI:

We have a container GUI that has many Widgets together as a 3drv tool.

April PIXON/Amos Meeting

Minutes

Question 1: Background, Poisson, Gaussian

Conclusion: Amos assumes Gaussian distribution noise for Poisson is Gaussian in the upper limit and low signal Poisson regions are less important. We will avoid negative signals (either by undersubtracting background or using a pedestal.)

(Amos) 2 primary questions for his new release. Is the data background dominated (signal mostly sigma) or Poisson noise.
(us) Depends on whether it is polarized or not
(Amos) We need to ignore negative signals
(Us) If we pedestal, we don't do it pixel by pixel but as a smooth or flat value
(Amos) So our sigmas are not Poisson distributed?
(Us) No, sigma is still sqrt(sum[inputs])
(Amos) With very low counts, we must be careful of Poisson
(Us) Aren't you always minimizing chisq?
(Amos) Yes, chisqr = (obs-predicted)/sigma^2, is sigma from prediction or actual counts?
(Us) actual counts
(Amos) So I don't have to worry about Poisson statistics even though chisq is bad for Poisson at low counts. Since we're dealing with total counts of 100 or more we're okay.
(Us) Yes, low count regions (~10s) are less important. And we don't have many single digit pixels-- they won't drive the solution, right?
(Amos) If that's total density, you're screwed [paraphrasing heavily here]
(Us) It's not.
(Amos) So we're dealing with Gaussian noise.
(Us) We prefer the Poisson switch.
(Amos) No, it deals with it in a different way.
(Us) So the Poisson switch was meant to be used only when the signficant areas are low signal?
(Amos) Yes. So let's work under the assumption that the signal is Gaussian.
(Us) Even if large areas are insignificant <u>but</u> low signal?
(Amos) If you have a Poisson distribution, don't background subtract.
(Us) Isn't a Poisson switch a Gaussian at high signal?

(Amos) No, I use a different numeric process for P vs G. For now, we'll assume we have Gaussian distributed noise.
(Us) Okay

Question 2: Are polarization angles simultaneous, same detector, same boresight? If not, there's a problem.
Conclusion: Nearly, yes, and yes. So not a problem.

Question 3: Kernels and Memory ('Original Agenda' item 1)
Conclusion: The new version will calculate kernels on the fly.

(Amos) instead of doing kernel smoothing, assumes a Gaussian (or parabola, etc) and approximates it with a triple smoothing of 3 square wave smoothings of the same amount, to speed it up.

- faster
- doesn't require storage

He will implement this in the new version.

Question 4: Finding voxel lines of sight ('Original Agenda' item 2)
Conclusion: New version will switch to Voxel-based

Amos wants to do John's request to enable the projection and back projection as a standalone renderer. In doing that, is now veering towards a voxel-based renderer instead of the current pixel-based ones.
John worries about performance, Amos says he can just code cleverly.
For a given input e- density to fit, need to multiply by an emissibity for each view and each polarization. Amos doesn't want to write this so we must. We discussed whether to do it as a subroutine or store it as an array.
Storage = # voxels * # views * # polarizations
(Us) Paul & Jeff wrote this 4 years ago. The routines do emissibity on the fly and it was not a bottleneck.
(Jeff) It's a quick hop over planes, 256 loops of a 256x256 plane.
(Amos) So I can leave it as a subroutine that you'll supply. So I'm going to write a projection operator that inputs an electron density that calls the routine that calculates the emissivities.
(Jeff) Just like the old way of replacing DSF and DSFT
(John) It's a done thing, yes?

(Amos) Yes.

Question 5: Paul's Geometry (Original Agenda item 3)
Conclusion: (later) moot, as it's only for creating models, yay!

Question 6: Different Nd (datum), Nv (voxels) (Original Agenda item 4)
Conclusion: All data must have same Nd, image Nv will be either Nd or 2xNd, goal is Nd=512^3

(John) We'd be thrilled with 1024^3, satisfied with 512^3.
(Amos) Main bottleneck: forward and backward projection
(All) Get it to work at 512^3 then speed it up later.
(Sandy) It needs to fit into 4GB (or 8GB tops)
(John) Should we assume 64-bit
(Sandy) Yes, assume 64-bit, 4GB Ram.

(John) We will always collapse 2048x2048 to 1024x1024.
(Amos) If Nd does not equal Nv, the pixel size and voxel size differ, that's a problem. Take a voxel and project it to the FPA, does it encounter 1 pixel, many, or a fraction?
(John) Is idea Nd = 2 * Nv? Is Nd always <= Nv?
(Sandy) This is 2 issues: is it right, and can it be done?
(Amos) If Nd does not equal Nv the rendering computation is affected.
(all) At same or factor of 2 resolution is all we need.
[note: it's implicit that Nd1 = Nd2, but not discussed]

Question 7: Licensing (Original Agenda item 5)
Conclusion: We will direct them to Amos and they would get the license.

Original Agenda

Sandy Antunes
March 16, 2007

Since we are committed to 'classic' (cartesian) Pixon, that is the code base I am working with. Right now I can do N=256 on a standard (2GB single processor) desktop, and I

now have access to test an N=512 run on a 16GB quad-core machine.

Topic-wise, the issues I would like to work on are:

1) kernels and memory
2) LOS map
3) Paul's geometry
4) when Ncube != Ndata
5) licenses and the community

1) Improving performance of the kernel functions, either by:

   a) pre-allocating a single initial memory chunk then generating kernels on the fly (low memory usage, high processor usage)

   and/or b) using 'assoc' for kernels

   Both of these primarily deal with pxnkern__init and pxnkern__get.

2) Extracting a cube where the value for each voxel is simply the number of lines of sight to data for that voxel. As you state that's easy, it'd be nice to spend an hour or so with you to extract this as a simple, meaningful image data item that we can look at with other tools.

3) If possible, take a look at Paul's geometry code, as he notes himself it's not the best approach. It's in our routine of center_data_07.pro, and as Paul's comments indicate:

```
; ??? this subroutine just slides the data array around on the data plane
; to get the center of the image cube onto the center of the data array.
; It also ensures the sun center (which may be diff from optical
; center) is transformed identically.
;
;(Paul wrote) I think what we should be doing is keeping the optical center
; in the center of the data array (sr.k0 = [1,1]*(N-1)/2 for all frames)
; and altering the pointing (sr.z0, sr.eps, sr.set)
```

So investigating how to shift to Paul's 2nd approach would be helpful.

4) Pixon has hooks as if it can support data N being different from image N, but I haven't been able to get that to work. So currently our image cube N^3 is set by our data N^2 (in classic pixon). But it's possibly I'm misreading the documentation/code.

Is the 'different N' code working to your knowledge, or is this something I should work on?

Method for Rotational Tomography

Sandy Antunes, Jan 2007

(HAE=Heliospheric Aries Ecliptic, Carr=Carrington Heliographic Coordinates)

Currently, we ingest FITS coordinates and convert to HAE for the reconstruction. This means we typically neglect times, since HAE is an absolute reference frame. As a corollary, this also means we assume the user has selected observations in roughly the same time frame so as to minimize feature evolution!

ROTATIONAL RECIPE:

Have N_a views at time t_a with coordinates in HAE
Have N_b views at time t_b with coordinates in HAE

During time (t_a-t_b), the sun rotated 13.1988 degrees/day, in
a solar rotational frame, this is dR=0, dZ=0, d(phi)=alpha;
alpha = 13.1988 degrees * (t_a-t_b in days)

So convert N_b(HAE) -> Nb(Carr)
= r, theta, phi

Add in tomo rotation 'alpha'
phi' = phi + alpha
r'=r
theta'=theta

Convert back:
Nb(Carr)' = N_b(HAE)'

Continue as usual.

Pixon Test Script

Here is a simple test case for Pixon, easily run with '*@p_run_test*'. It is in dev/pixon/tests. It creates a simple fluxrope (default resolution N=32), renders it from two angles, then attempts to reconstruct it with classic Pixon.

```
;; extremely simple test case for Pixon, create and render a fluxrope

  .reset
  on_error,2
  N=32
 nviews=2

  modelname='fluxrope'
  timage=pixonmodels(N,modelname,pxwrapper,stuff,stuff1)

 prep_3drv,pxwrapper,views=nviews,N=N


; the following lines let you change memory and performance settings.

 pxwrapper.sr.interp=5; 5 = use C version
 pxwrapper.sr.interp=0; 0 = force native IDL for big runs

 pxwrapper.sp.noshm=1; 1 = do not use shm
 pxwrapper.sp.noshm=0; 0= use shm
 pxwrapper.sp.noshm=-1; -1 = sometimes use shm

 noct=2; half as many, check memory usage for large N
 noct=4 ; default, will be set to 4 in Pixon

 npo=1; half as many, check memory usage for large N
 npo=2 ; default, will be set to 2 in Pixon
 pxwrapper.sp=p_tweak_kernel(pxwrapper.sp,noct,npo,/debug)


  p_generate_pixon,'classic',pxwrapper,oPixon,image=timage
```

```
   datum=stereo_project(pxwrapper,oPixon)
   tv_multi,datum,res=6,/axes,/border
   p_insert_data,pxwrapper,datum,oPixon

;   oPixon->Full,/restart ; works

   p_pixonloop,pxwrapper,oPixon,2,8,/init,/full

   p_report,pxwrapper,oPixon,/save,/plot,/echo

; global constants to set for physically real problems:
;sr.n0, sr.chl, sr.L0, sr.lla0, sr.d, sr.interp, sr.psp
;sp.reltol, sp.abstol, sp.mxit, sp.npo
```

Tests: Pixon Count Due to Signal, Resolution, n0

April 2007 & June 2007

Use of n0 and MSB

MSB, n0, and Pixon Counts

Conclusion 1: PIXON requires that:max(data)>>n0 and also max(data)>1.

Conclusion 2: For identical data sets differing only by the scaling of the data signal alters the Pixon count but not the answer:

larger max(datum) -> higher Pixon count,
higher Q
higher chi2/DOF

Thus you get more Pixons for a larger max(datum).

Altering 'n0' does not alter the convergence or Pixon count or Q, nor the fact that the solution image and renders retain their appropriate scaling. Since n0 is determined by the data and instrument, we therefore can safely rescale our data to maximize our desired Pixon 'headspace' so long as we rescale n0 by the same factor.

Any rescaling should occur before the 'p_generate' generation of the Pixon object, because the scaling factor n0 is used to calculate the Istd and flimb functions.

For real data in MSB, n0 is the conversion factor to photons, e.g. Cor 2 is typically Bsun/DN=6.5e-11 and DN/photons=15 so n0=6.5e-11 * 15 =~ 1e-9. The real data ends up being around <= e-6. So we need to rescale, for example by 1e+10 so data <= e+4 and n0=1e1.

Tests:

Tests: 'fails' = "Warning, Pixon count LE 0'

Used mxit=20, one iteration

1) n0=1
no data or image rescaling
max(input image) =~ 1E10
max(datum) =~ 1E09

max(solution image) =~ 9E09
max(render) =~ 2.5E09
Q =~ 1E13, chi2/DOF =~ 1E12
Pixon count = 12695

2) n0=max(timage) =~ 1E10
data and image rescaled by n0
max(input image) = 1
max(datum) =~ 0.255

FAILS

3) n0=max(datum) =~ 1E9
data and image rescaled by n0
max(input image) =~ 3
max(datum) = 1

FAILS

4) n0=1E7 (an arbitrary value less than image or datum max)
data and image rescaled by n0
max(input image) =~ 1000
max(datum) =~ 250

max(solution image) =~ 593
max(render) =~ 260

Pixon count = 363

5) n0=1E7 (an arbitrary value less than image or datum max)
no data or image rescaling
max(input image) =~ 1E10

max(datum) =~ 1E09

max(solution image) =~ 9E09
max(render) =~ 2.5E09
Q =~ 1E13, chi2/DOF =~ 1E12
Pixon count = 12695

i.e. same result as case 1)

6) n0=1E4 (an arbitrary value less than image or datum max)
data and image rescaled by n0
max(input image) =~ 1E6
max(datum) =~ 2.5E5

max(solution image) =~ 9E5
max(render) =~ 2.5E5
Q =~ 5E5, chi2/DOF =~ 2E4
Pixon count = 10365

6) n0=1E4 (an arbitrary value less than image or datum max)
no data and image rescaling
max(input image) =~ 1E10
max(datum) =~ 1E9

max(solution image) =~ 9E9
max(render) =~ 2.5E9
Q =~ 5E13, chi2/DOF =~ 1.5E12
Pixon count = 12695

Sigma Tests and Cautionary

sigma=1 works, but sigma=array of 1s does not.

Note, to avoid data size errors do not use raw arrays, use pointers.
Bad:
opixon->set,sigma=[N,N,Nd] or opixon->replace,sigma=[N,N,Nd]

e.g. if sigma=sqrt(datum) for Nd=2 and you have

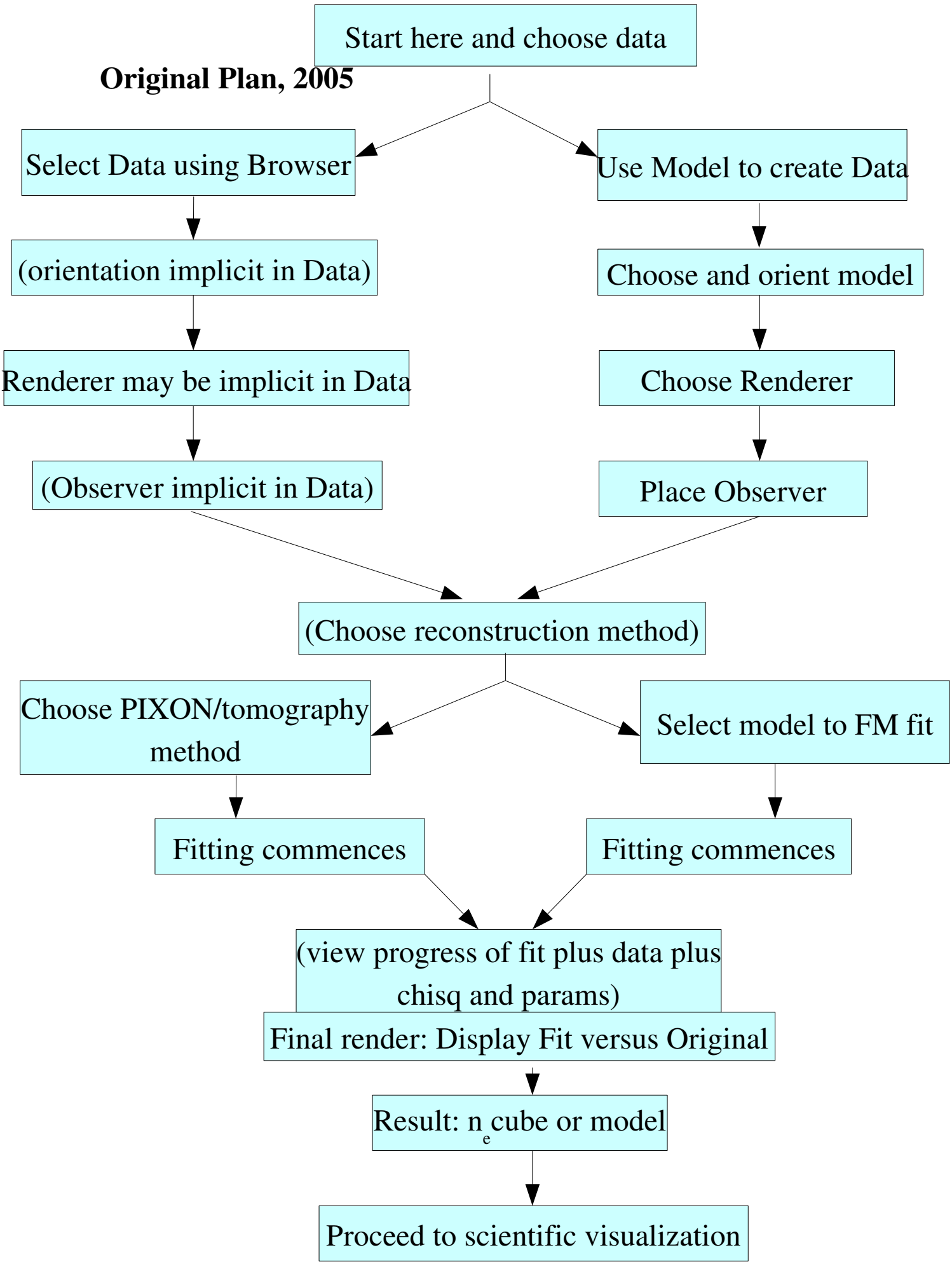
```
sigma=[N,N,2]
opixon->set,sigma=sigma
```

actually messed it up:

```
opixon->get(/sigma,/nop)   returns  [N,N,2,2]
```

So always cast to pointers, e.g.

```
opixon->replace,sigma=sigmaptrs
or opixon->replace,sigma=force_ptrs(sigma)
```


Original Plan, 2005
Start here and choose data
Select Data using Browser
Use Model to create Data
(orientation implicit in Data)
Choose and orient model
Renderer may be implicit in Data
Choose Renderer
(Observer implicit in Data)
Place Observer
(Choose reconstruction method)
Choose PIXON/tomography method
Select model to FM fit
Fitting commences
Fitting commences
(view progress of fit plus data plus chisq and params)
Final render: Display Fit versus Original
Result: n e cube or model
Proceed to scientific visualization